UBIQUEST, For Rapid Prototyping of Networking Applications

Ahmad Ahmad Kassem, Christophe Bobineau, Christine Collet, Etienne Dublé, Stéphane Grumbach, Fuda Ma, Lourdes Martinez, Stéphane Ubéda

UBIQUEST, For Rapid Prototyping of Networking Applications

Ahmad Ahmad-Kassem[3], Christophe Bobineau[2], Christine Collet[2], Etienne Dublé[1],
Stéphane Grumbach[3], Fuda Ma[4], Lourdes Martinez[2], Stéphane Ubéda[3]
[1]CNRS, [2]Grenoble Institute of Technology, [3]INRIA, [4]INSA-Lyon
{Christophe.Bobineau, Christine.Collet}@grenoble-inp.fr
fuda.ma@insa-lyon.fr
{etienne.duble,Lourdes-Angelica.Martinez-Medina}@imag.fr,
{ahmad.ahmad_kassem, Stephane.Grumbach,stephane.ubeda}@inria.fr

ABSTRACT
An UBIQUEST system provides a high level programming abstraction for rapid prototyping of heterogeneous and distributed applications in a dynamic environment. Such a system is perceived as a distributed database and the applications interact through declarative queries including declarative networking programs (e.g. routing) and/or specific data-oriented distributed algorithms (e.g. distributed join). Case-Based Reasoning is used for optimization of distributed queries when as there is no prior knowledge on data (sources) in networking applications, and certainly no related metadata such as data statistics.

Categories and Subject Descriptors
H.2 DATABASE MANAGEMENT [**Languages,** Systems and Software]: *Query languages*, *Query optimisation and processing*, *Rule-based program execution,* Distributed databases, Distributed systems, *Reasoning,* Information *networks*

General Terms
Your general terms must be any of the following 16 designated terms: Algorithms, Management, Measurement, Documentation, Performance, Design, Economics, Reliability, Experimentation, Security, Human Factors, Standardization, Languages, Theory, Legal Aspects, Verification.

Keywords
Declarative networking, programming abstraction, case-based distributed query optimization.

1. INTRODUCTION

The trend towards ubiquitous computing is accelerated with – particularly, wireless networking – technologies interconnecting an increasing number of heterogeneous (mobile and wearable, energy constrained, personalized) devices that generate large amounts of data. These devices are autonomous, either, static or mobile and present constraints such as energy or communication capabilities. They usually take part in dedicated ad hoc networks, where applications deployment, configuration and management are tedious and require significant human involvement and expert knowledge.

IDEAS12 2012, August 8-10, Prague [Czech Republic]
Editors: Bipin C. Desai, Jaroslav Pokorny, Jorge Bernardino

In [1] we introduce our vision of a new high-level programming abstraction based on the emerging and promising declarative networking approach and declarative data manipulation expressions. Declarative networking is an emerging data-centric approach where the distributed environment is perceived as a distributed database and the applications interact through declarative queries [19, 17, 16]. This approach has been pursued at the network layer with the use of recursive query languages initially proposed to express communication network algorithms such as routing protocols [17] and declarative overlays [16]. It has been further pursued in [15], where execution techniques for Datalog are proposed. Distributed query languages thus provide new means to express complex network problems such as node discovery [22], route finding, path maintenance with quality of service [4], topology discovery, including physical topology [3], etc. The declarative networking approach is well-adapted to social systems (e.g. games, social networks, sharing), where data is pushed or pulled with incomplete knowledge in a dynamic environment.

Also declarative query languages have already been used in the context of ad-hoc networks. Several systems for sensor networks, such as TinyDB [18] or Cougar [8] have been proposed. They use the relational model to represent device (sensor) features and application data; they offer SQL-like languages to express data manipulation. These systems also address solutions to perform efficient data dissemination and query processing. In both cases, a distributed query execution plan is computed in a centralized manner considering the network topology and the capacities of the constrained nodes, which optimizes the placement of sub-queries in the network [8, 18]. Declarative methods have been used also for unreliable data cleaning based on spatial and temporal characteristics of sensor data [14].

As far as we know there is no system that integrates in a uniform way, network aspects, middleware and data management. UBIQUEST merges declarative programing languages and query languages for specifying data manipulations and distributed algorithms. Furthermore, these languages are used to intentionally express the destinations of messages, for naming and accessing data in the context of networks and dynamic environments.

The work presented in this paper describes the architecture and components of an UBIQUEST system – http://ubiquest.imag.fr – that implements this approach.

It is a sort of a large distributed database system that provides a unified view of "objects" handled in networks and applications. It blurs the borders between network, operating system and middleware layers. However, from the data management point of view it should provide a means (i) to localize data (mobile applications) or define the scope of a query, (ii) to consume, filter

and aggregate data (continuous queries), (iii) to consider query operators that may correspond to programs, (iv) to optimize query even when no metadata or statistics are available. For that, we use Case-Based Reasoning (CBR) – learning the cost of query plans (cases) while executing them – and pseudo-random query plan generation when classical optimization techniques are inappropriate.

The paper is organized as follows. Section 2 gives an overview of our approach based on an example unifying data and networks management functions. It defines an UBIQUEST system as a set of interconnected UBIQUEST nodes. Section 3 presents the architecture of an UBIQUEST node and details its components. Section 4 focuses on the execution engines that perform program execution and global query evaluation. Section 5 presents our proof of concepts as a platform for simulating UBIQUEST systems. Finally, Section 6 concludes the paper and discusses future work.

2. UBIQUEST DATA-CENTRIC APPROACH

With declarative networking, the network is abstracted as a large distributed database providing unified view of "objects" handled by both networks and applications. Such a database stores information about the declarative programs, routers configuration, states and characteristics of the network. Rule-based programs usually correspond to network operations or protocols triggered by data updates. Rules are evaluated over local data and may communicate updates to other nodes in the network using communication primitives.

The UBIQUEST approach merges the strengths of two areas (i) databases, and (ii) declarative networking. With this approach a programmer can specify the behaviour of the system / application (the what) rather than having to describe the details of the system (the how). This allows going one step further in the overlapping approaches, for example with destinations of messages resulting of a query.

An UBIQUEST system runs on a set of computing devices interconnected through a wireless network (cf. Fig. 1). Every device embeds a virtual machine in charge of data management, processing queries (data selection and updates) and messages propagation. A message is the unit of communication among UBIQUEST nodes. It has two main parts: (i) *networking information* (e.g. logical destination, next hop, TTL) and (ii) a *payload* where the content of the message (i.e. queries or items) is embedded.

All exchanges between nodes related to communication protocols, to resource discovery or to any other applicative aspects are carried out by queries and data. This blurs the traditional distinction between communication middleware and application layers. Queries are defined using either rule-based languages (e.g. for network data query expressions or distributed algorithms) or declarative query languages (e.g. for querying application data with a global point of view). For a detailed presentation of these languages refer to [1].

Query optimization is based on CBR-based approach and pseudo-random query plan generation. This means that we learn the cost of query plans (cases) while executing them. These cases are reused for generating plans for further similar queries. If there is no convenient case, we use classical heuristics and random choice (e.g. when there is no statistics for join ordering and selection of algorithms) to generate query plans.

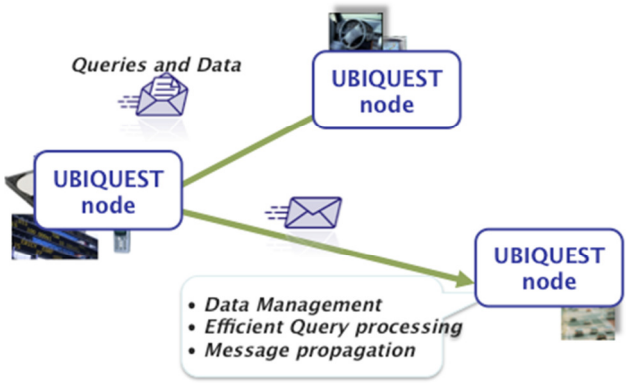


Figure 1. An UBIQUEST System

To illustrate our approach, let us consider an application concerning a virtual world game divided in areas and having some avatars that are located within a single area at a time (see Fig. 2). The objective of the game may be social interaction or team fighting; this does not matters for understanding the example. Every node of an UBIQUEST system has information on its own avatars and their neighbors (avatars located in the same area). Data location is thus application driven.

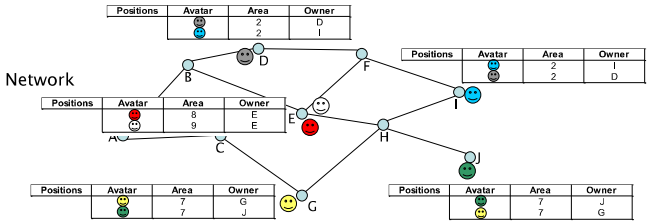


Figure 2. Application scenario

Such information is stored in an *Itemset* data structure of type:

Positions*(Avatar avatar{key}, Int Area, NodeID owner)*

For example, at node *D* the **Position** itemset has two items:

Positions	Avatar	Area	Owner
		2	D
		2	I

showing that the node is the owner of the *Grey* avatar which is in the area 2; also the *Blue* avatar is in the same area but owned by node *I*. The **Positions** table at node I contains the same tuples (Fig. 2).

Such a **Positions** table is actually a fragment of a virtual table that maintains the information on all the avatars, giving for each of them, the area in which it is and its owner node. The virtual global **Positions** table for our application example is:

Positions	Avatar	Area	Owner
		7	J
		7	G
		8	E
		2	I
		2	D
		9	E

Considering this global view, the query to select all avatars in the virtual world and the zone where they are located is:

SELECT Area, Avatar FROM Positions;

This query can be posed at any node and the system will globally execute it. Alternatively, for restricting the scope of the query at a node level, one has to use the keyword *LOCAL* indicating that the query has to be evaluated over local data only.

Let us now assume that the *Yellow* avatar, owned by node G, is moved from area 7 (where avatar *Green* owned by node J is localized) to area 8 where the *Red* avatar (node E) is localized. The **Position**s table after this operation follows:

Positions	Avatar	Area	Owner
		7	J
		8	**G**
		8	E
		2	I
		2	D
		9	E

The movement is coded by several updates executed at node G (owner of *Yellow*) for cleaning area 7, changing the Area attributes of the avatar and finally for storing the new area exploration. The first update for cleaning the area 7 is:

Delete from Positions
*Where Area = (**Local** Select Area from Positions*
where Avatar = 'Yellow')
and Area not in (Local Select Area from Positions
where Avatar <> 'Yellow' and Owner = SELF)
***Stored on SELF**;*

The keyword *LOCAL* indicates that the subquery has to be evaluated by the node over local data only. The sub-queries are local and the delete operation too as it concerns only data stored on *SELF*. Such a query is executed at the node level and processed in a distributed way with the following principles:

1. No centralized control. Query processing is performed in an environment that is highly dynamic, and has to adapt to and recover from the network evolution. The control needs to be fully distributed over the network.
2. Scarce metadata. The network being highly dynamic, there is no stable knowledge on the data organization. Resource discovery is combined with networking protocols.
3. Everything in the database. The network management is done through queries.

3. UBIQUEST NODE

An UBIQUEST node is a device equipped with an UBIQUEST Virtual Machine (UBIQUEST VM) complemented with a Device wrapper that allows device/VM interaction (see Fig 3). The UBIQUEST VM is composed of: (i) a Local DMS, (ii) an UBIQUEST API, and (iii) an UBIQUEST Engine comprising sub-engines in charge of evaluating global queries, executing rule-based programs, maintaining sensed data and the list of physical neighboring nodes.

3.1 Local DMS and UBIQUEST API

The Local DMS stores and manages data as Itemsets: application data (e.g. sensed data), network data (e.g. routing tables, neighbor table), rule-programs (e.g. distributed algorithms that can be dynamically loaded/removed to/from the system), and internal data (e.g. device specific data) used for running other UBIQUEST VM components.

The UBIQUEST API manages all interactions between the UBIQUEST Engines and the rest of the world: local applications, device sensors and other UBIQUEST VM through message exchange.

As shown in Fig. 3, the API is composed of: (I) the *Application API*, in charge of the interaction with applications running on the local node, (ii) the *Reception* and *Emission* modules to deal with message exchange among UBIQUEST nodes, (iii) the *Sensing API* that locally stores data coming from sensors embedded in the

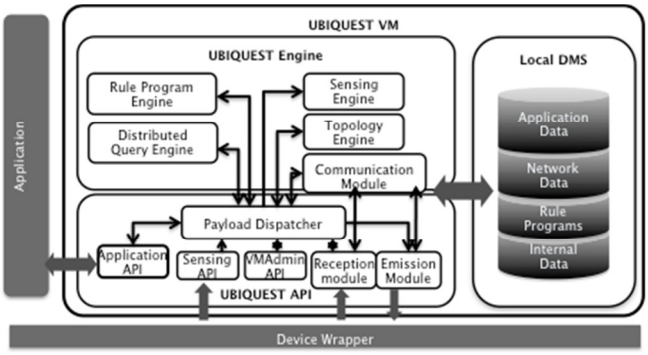


Figure 3. UBIQUEST node components

physical device, and (iv) *the Payload Dispatcher*, which manages *Payload* exchange among UBIQUEST VM sub-components.

The *Application API* module validates DLAQL queries/updates submitted by applications, and translates them into an internal representation before sending them to the *UBIQUEST Engine* for evaluation. The *Reception Module* receives messages from other UBIQUEST nodes and decides if the payload of the incoming message has to be treated locally. It checks if the local node is part of the *logical destination* of the message. This process may involve interaction with the *UBIQUEST Engine* to resolve intentional expressions of *logical destinations* (i.e. destination expressed using a query). Finally, the *Reception Module* sends the *Payload* of the message to the *Payload Dispatcher* to treat it, if the local node is one of the destinations, or forward the message to other destinations through the *Emission Module,* if not.

Using *Payload*, *logical destinations* and a *ProgramId* identifying a dissemination protocol, the *Emission Module* builds a new message, invokes the UNIQUEST Engine to compute the immediate physical destination(s) from the logical ones, and sends the message over the network.

The *Payload Dispatcher* maintains a record of the identifiers of payloads that are currently executed at the node. This allows determining if a received payload was already executed, and thus avoids loops. When it receives payloads from the *Application API*, it generates a new identifier for registering. When it receives payloads from the *Reception Module*, the *Payload* is forwarded to the *UBIQUEST Engine* for treatment.

When a payload is not in the record, the *Payload Dispatcher* generates a new identifier, registers the payload and transfers it to the corresponding engine. If the identifier is in the records, the payload dispatcher transfers it to the corresponding engine instance according to the query/result type and the payload identifier. If the message contains several destinations, the payload dispatcher sends it to the emission module, which constructs a message and propagates it over the network using a program (dissemination protocol) selected by the *UBIQUEST Engine* (*Communication Module*).

3.2 Sensing and Topology Engines

These two modules are autonomous and react to changes in the environment detected by the device and signaled through the *Device Wrapper*. The *Sensing Engine* gets the measures coming from physical sensors embedded in the device (e.g. temperature, location) and stores these values in corresponding itemsets. These itemsets are predefined and adopt a common structure (e.g. itemset ***Temperature****(NodeId {key}, value)*). The Topology Engine is responsible of updating the *Link* itemset, defined as

Link*(NodeId {key}, Neighbor {key})* according to physical network connections that are established or removed. The Link itemset is mandatory and is sufficient to permit communication among nodes.

3.3 Communication Module

The *Communication Module* has two different roles: (i) determine if the local node is part of the *logical destination* of incoming messages, and (ii) determine what is(are) the next hop(s) to transmit a message to a *logical destination*.

The *logical destination* of a message is either expressed *extensionally* using a list of node identifiers, either expressed *intentionally* using a query returning node identifiers, or expressed by a combination of both. If it is expressed extensionally, determining if the local node takes part in the logical destination of a message is straightforward. In the other case, the *Communication Module* asks the *Distributed Query Engine* to solve the intentional destination (i.e. obtain extensional destinations) before deciding.

To determine the next hop(s) for propagating a message, the *Communication Module* selects a *propagation program* and invokes the *Rule Program Engine* to execute it. The default *propagation program* simply do broadcasting to all neighbors (i.e. the next hops correspond to all items of the Link itemset). Other *propagation program*s may be written by developers (e.g. by exploiting and maintaining a routing table) and may be automatically selected by the *Communication Module*.

3.4 DQE Engine

The *Distributed Query Engine* is responsible of executing global – DLAQL queries. The DLAQL language extends the well-known SQL2 data manipulation language to conform to the data distribution policy of UBIQUEST. This means that a DLAQL expression may explicitly indicate on which UBIQUEST node data has to be stored. The role of the *DQE Engine* is to build and execute efficient local query execution plans according to a given cost function (expressed as a combination of real cost parameters). Execution plans are composed of classical physical operators (implementing algebraic operations) and specific operators to invoke program or propagate subqueries. Efficient execution plans are selected using a combination of Case-Base Reasoning and pseudo-random query plan generation.

The *Distributed Query Engine* is composed of: (i) a *Query Scheduler*, (ii) a *Query Optimizer* and (iii) an *Execution Engine*. The *Query Scheduler* rewrites a global query into a set of sub-queries and schedules their evaluation (e.g. a global *UPDATE* query is decomposed into a sequence of *SELECT*, *DELETE* and *INSERT* sub-queries to read the old value, delete it and insert the new value). Moreover, this module rewrites a query considering local and distant *Itemset* fragments generating a query (or set of queries) equivalent to the original one.

The *Query Optimizer* is based on the Case-Based Reasoning (CBR) approach as in [22]. It proposes to retrieve and adapt query plans using the experiences gained from the execution of past similar queries. When no knowledge is available it randomly generates query plans using classical heuristics [13, 23].

3.5 Rule Program Engines

The *Rule Program Engine* is in charge of executing rule-based declarative programs exploited for specifying distributed algorithms (e.g. networking protocols, sub-query execution). The engine selects which rules have to be triggered and execute them over the local data. The rule execution may involve local data storage or emission to neighborhood.

The proposed Netlog and Questlog rule-based programing languages extend *Datalog* with communication primitives, as well as aggregation and non-deterministic constructs which are standard in network applications. The computation of rule programs is local, and the result can be either stored locally on an UBIQUEST node on which the rules run, or sent to other nodes.

The *Rule Program Engine* receives payloads from the *Payload Dispatcher* and has to treat their *Contents* containing either items (new facts or query results) or predicates corresponding to queries.

If a *Content* contains facts, the *Rule Program Engine* identifies which rule-program has to be triggered by comparing new facts with predicates in the rule body (*Netlog*). Then, it retrieves the corresponding rules from the DMS and evaluates them in forward chaining mode till a fix point is reached.

If a *Content* contains a predicate corresponding to a query (*Questlog*), the *Rule Program Engine* identifies which rule-program has to be triggered by comparing the predicate with rule head, then it retrieves the corresponding rules from the DMS and evaluates them in backward chaining till the full query result is computed.

If a *Content* contains query results, these results are exploited to continue query evaluation.

4. Query and rule-based program execution

4.1 Distributed Query Execution

As said in section 3.4, incoming global queries are rewritten by the *Scheduler* as a sequence of queries that are to be evaluated in correct order to produce the final result (e.g. evaluate asynchronous subqueries before executing rewritten upper level query). These queries generally contain access to local data and to distant data, according to the horizontal fragmentation of global itemsets. An optimal query plan has to be generated for each one of these queries.

A query plan is a tree whose nodes are physical operators corresponding to data manipulation. A root node corresponds to a DLAQL command (i.e. *SELECT*, *INSERT*, *DELETE*, *UPDATE*), intermediate nodes correspond to computation operators (e.g. unions, joins, filter or aggregate), and leafs correspond to data access operators: local DMS querying, sub-query emission to all neighbors, or rule-based program invocation.

A case is generated for any (sub-)query expression Q evaluated on a node. It is composed of the expression of Q, a query plan P for Q and a set of cost parameter measures taken during the execution of P. A case is stored in the local case base of the node. The following example is a case for a query finding all avatars in area 7:

{Q = "Select Avatar from Positions where Area = 7",
P = Union(***DMS****(*$\sigma_{Area=7}$*(Positions)),*
SubQ*(*$\sigma_{Area=7}$*(Positions))),*
Cost = {Energy=0.5%, Time=12ms, Memory=2%, …}
}

The query plan P involves computation of a subquery on local DMS (*DMS* operator) and emission of a global sub-query (*SubQ* operator). Cost parameters are normalized when possible.
In our approach, the query plan generation is a recursive process.

The optimizer checks if the incoming query Q is known in the case base, i.e. if there exist similar cases corresponding to this query, select the best case/plan among them according to the optimization objectives (i.e. minimizing the given cost function). A similarity function to compare a case and a query has been defined based on classification of query expressions (i.e. based on DLAQL clauses, see [20]). If a plan is selected then it has to be adapted to the current Q expression and to be executed, otherwise a plan is generated using a pseudo-random approach. This generation process applies classical heuristics and random choice when missing metadata (e.g. statistics) is needed. For example, if Q contains a set of join operations, it select randomly one join operation and a corresponding algorithm (join operator) and return a plan for this join operation involving two sub-queries. The same process is applied recursively for every sub-query. An in depth description of the optimization process is given in [20].

The Execution Engine executes a query plan P using the well-known Iterator model [10] for the physical operators. It also coordinates the local and distant sub-queries and constructs a final result from sub-query results. During the execution, the cost parameters (energy, time, memory etc.) are measured and a new case is built.

4.2 Rule program execution

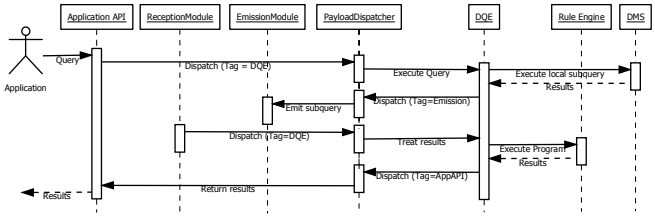
As we already explained the *Rule Program Engine* receives payloads from the *Payload Dispatcher* and has to treat their *Contents* containing either items (new facts or query results) or predicates corresponding to queries.

In addition, the *Rule Program Engine* propagates new items or new queries to other nodes, through the UBIQUEST API, and/or stores new items in the DMS. The *Engine* has some additional functions, such as timers, necessary for networking protocols, and also uses optimization techniques, such as the triggering of rules by new facts, which avoid unnecessary computations, when there are no changes in the input of rules.

Let us assume the following simple routing table maintenance protocol:

↕Route(SELF, dest, dest, 1) :- Link(SELF, dest).
↕Route(SELF, dest, neigh, l2) :- Link(SELF, neigh),
Route(neigh, dest, _, l1), l2 := l1 + 1.

These rules will be executed if a new neighbor is discovered (i.e. new item in the *Link* itemset). Satisfied rules involve local storage and broadcasting of new facts (heads of the rules) to all neighbors (↕ symbol). This is done via the *Emission Module* alone as the destination is expressed extensionally (i.e. *all neighbors*). The following sequence diagram describes the whole process:



4.3 Combining DLAQL queries and rule programs

Going back to the evaluation of the query

Q = "Select Avatar from Positions where Area = 7", as

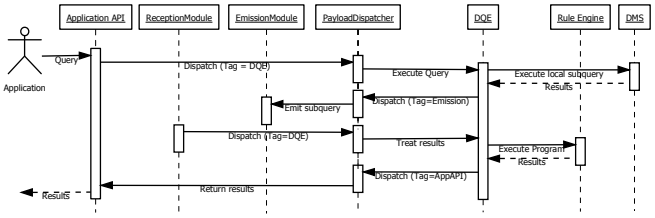
*Union(**DMS**($\sigma_{area=7}$(Positions)),*
***SubQ**($\sigma_{area=7}$(Positions)));*

One can figure out, this leads to useless sub-query evaluation. The distant part (*SubQ*) is useless because the node locally stores its avatar (localized in area 7) and its neighbors (i.e. in the same area).

Such knowledge can be represented as rule-based programs executing specific algorithms to solve these sub-queries. Here, this program is the local identity where no sub-query is emitted:

↑Positions(_, area, _) ← Positions(_, area, _).

The DQE Engine may exploit this program to solve (part of) the query Q, thanks to the correspondence table between queries and programs. The following sequence diagram shows the interactions among UBIQUEST node components for such an evaluation.



5. SIMULATION PLATFORM

We develop a platform (see Fig. 6) facilitating the development and monitoring of UBIQUEST applications. This platform offers tools for editing and compiling rule-based programs, a allows the simulation of network of UBIQUEST nodes.

The simulation platform has three main components:

- The *Network Editor*, which allows to build and simulate a network with various UBIQUEST nodes;
- The *Network Monitor*, which allows to visualize and interact with the network at run time; and
- The Node Monitor, which allows to monitor the activity of and interact with individual nodes.

The *Network Editor* allows creating groups of nodes, displaying the status of the nodes in each group and installing rule programs on them. They can have different colors, radio range, and characteristics, such as mobile or fixed. The system creates the groups and displays the nodes on the left part of the screen. Each node is listed and for each node one can see its identifier, address, position and radio range.

The *Network Monitor* offers the view of the different groups of nodes, represented by different shapes and different colors, and the connections between them (if the nodes are located inside the radio range of another node). Each node has a unique identifier. The *Network Monitor* also allows to interact with the network, and to modify its configuration before starting or during the simulation, by moving nodes, changing their radio range, or deleting edges or nodes for instance.

The *Node Monitor* exhibits information about the node selected by the user, displayed on the right part of the screen. It contains six tabs: *Display itemset*, *Programs*, *Messages*, *API*, *DQE*, and *Statistics*. The *Display itemset* tab allows the user to choose an *Itemset* existing in the DMS of the node and to display the values of each items of this *Itemset*. It is important to notice that the content is updated on the fly. For example, you can choose to display the content of the *Itemset* "Route" to see all the routes contained on the selected node. The next tab simply displays which programs are installed on the node with the possibility for the user to enable or disable them on this node. The *Messages* tab

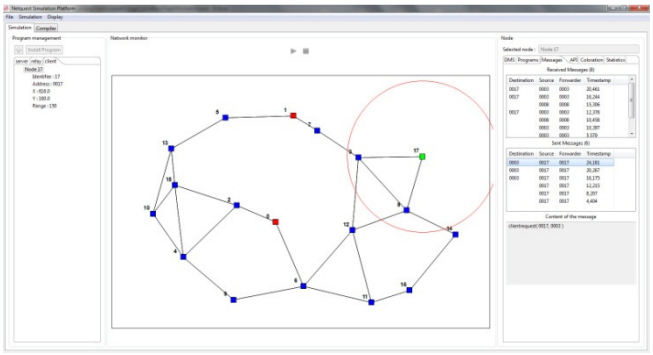


Figure 4. UBIQUEST Simulation Platform

displays all messages received or sent by the node. If you click on one message in particular, you can display its content. The *API* tab permits to modify the content of the DMS of the selected node by adding an item in one of the *Itemset*s of the node, or by submitting queries expressed in DLAQL, as an application would do. The *DQE* tab allows to monitor query execution by exploring the case base (i.e. query families, query plans and measures of computation cost), displaying the list of pending subqueries and partial results for any of the queries running on the selected node. The last tab shows some basic statistics about the node such as the number of Select queries or Update queries done in the database of the node.

We also developed a simulation and emulation environment for a detailed analysis and evaluation of queries for a large class of algorithms and protocols.

6. ACKNOWLEDGMENTS

This work has been supported by the ANR-09-BLAN-0131-01 UBIQUEST Project (http://ubiquest.imag.fr), financed by the French National Research Agency (ANR).

7. REFERENCES

[1] A. Ahmad-Kassem, C. Bobineau, C. Collet, E. Dublé, S. Grumbach, F. Ma, L. Martinez, S. Ubéda. A data-centric approach for networking applications. In: DATA, (2012), Rome, Italy.

[2] G. Alonso, E. Kranakis, C. Sawchuk, R. Wattenhofer and P. Widmayer. Probabilistic protocols for node discovery in ad hoc @multi-channel broadcast networks. In: Pierre, S., Barbeau, M., Kranakis, E. (eds.) ADHOC-NOW 2003. LNCS, vol. 2865, Springer, Heidelberg, 2003.

[3] Y. Bejerano, Y. Breitbart, M. Garofalakis and R. Rastogi. Physical topology discovery for large multi-subnet networks. In: INFOCOM, 2003.

[4] Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi and Alexander Sprintson. Algorithms for computing QoS paths with restoration, IEEE/ACM Transactions on Networking (TON), v.13 n.3, p.648-661, June 2005

[5] X. Chen, Y. Mao, Z. M. Mao, and J. Van der Merwe. Decor: Declarative network management and operation. SIGCOMM Comput. Commun. Rev., 40:61-66, January 2010.

[6] Z. Cheng, M. Perillo, and W. B. Heinzelman. General Network Lifetime and Cost Models for Evaluating Sensor Network Deployment Strategies. IEEE Transactions on Mobile Computing, 7(4):484-497, April 2008.

[7] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. Proc. VLDB Endow., 1:1153-1165, August 2008.

[8] A. J. Demers, J. Gehrke, R. Rajaraman, A. Trigoni, and Y. Yao. The cougar project: a work-in-progress report. SIGMOD Record, 32(4):53-59, 2003.

[9] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In Proceedings of the Thirtieth International conference on Very Large Data Bases. Volume 30 (VLDB '04), 588-599, 2004.

[10] G. Graefe. Query evaluation techniques for large databases. ACM Computing Surveys, vol. 25, Issue 2, June, 1993.

[11] S. Grumbach and F. Wang. NetLog, a rule-based language for distributed programming. In M. Carro and R. Pea, editors, PADL, volume 5937 of Lecture Notes in Computer Science, pages 88-103. Springer, 2010.

[12] W. Hoschek, F. J. Jaén-Martinez, A. Samar, H. Stockinger, and K. Stockinger. 2000. Data Management in an International Data Grid Project. In Proceedings of the First IEEE/ACM International Workshop on Grid Computing (GRID '00), Rajkumar Buyya and Mark Baker (Eds.). Springer-Verlag, London, UK, 77-90.

[13] Y. Ioannidis. Query optimization. ACM Comput. Surv., 28(1):121-123, 1996.

[14] S.R. Jeffery, G. Alonso, M. J. Franklin, W. Hong and J. Widom. Declarative support for sensor data cleaning. In: Fishkin, K.P., Schiele, B., Nixon, P., Quigley, A. (eds.) PERVASIVE 2006. LNCS, vol. 3968, pp. 83-100. Springer, Heidelberg, 2006.

[15] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, 2006.

[16] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In 20th ACM Symposium on Operating Systems Principles (SOSP), 2005.

[17] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In SIGCOMM, 2005.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. ACM Trans. Database Syst., 30(1), 2005.

[19] Y. Mao. On the declarativity of declarative networking. SIGOPS Oper. Syst. Rev., 43:19-24, January 2010.

[20] L. Martinez, C. Collet, C. Bobineau, E. Dublé, The QOL approach for optimizing distributed queries without complete knowledge. In: IDEAS, 2012.

[21] Microsoft, "Execution plan caching and reuse," [Online]. Available: http://technet.microsoft.com/en-us/library, 2008.

[22] M. Stillger, G. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001), Morgan Kaufmann Publishers Inc, 19-28, San Francisco, CA, USA, 2001..

[23] D. Subramanian and K. Subramanian. Query optimization in multidatabase systems. Distrib. Parallel Databases, 6(2):183-210, 1998.

[24] Y. Yu, D. Estrin, and R. Govindan, Geographical and Energy-Aware Routing: A Recursive Data Dissemination Protocol for Wireless Sensor Networks. UCLA Computer Science Department Technical Report, UCLA-CSD TR-01-0023, May 2001.